Natural Language and Information Retrieval
Degree in Data Science
ETSINF. Academic year 2023-2024

Practical Work 3

Session 1: Machine Learning Models, and Hyperparameters.

Reynier Ortega Bueno
rortega@prhlt.upv.es

Goal

The main objective is to train and validate several supervised classification models for addressing the tasks proposed in the shared task “Oppositional thinking analysis: Conspiracy theories vs critical thinking narratives”[1]. Firstly, students will practice supervised machine learning methods such as Support Vector Machines (SVMs), Logistic Regression, Decision Trees, and Multilayer Perceptrons (MLPs), applying them to real-world natural language processing problems. This hands-on experience will provide a solid understanding of how these models can tackle complex language challenges. Secondly, students will explore the benefits and drawbacks of combining multiple machine learning models. They will gain insights into ensemble techniques, including Voting, Stacking, Bagging, Boosting, and Random Forests (RF). By harnessing the collective strength of the simple models, ensembles can achieve better performance than using isolated models. Finally, the session introduces two methods for exploring the hyperparameter space, a crucial step in achieving optimal model performance. Python library scikit-learn will be used to implement and validate the models.

Bibliography

- DUDA, Richard O., et al. *Pattern Classification*. John Wiley & Sons, 2006. Chapter 6
- THEODORIDIS, Sergios; KOUTROUMBAS, Konstantinos. *Pattern Recognition*. Elsevier, 2006. Chapter 4
- BISHOP, Christopher M.; NASRABADI, Nasser M. *Pattern Recognition and Machine Learning*. New York: Springer, 2006. Chapters 4, 5 and 7
- FERNÁNDEZ, Alberto, et al. *Learning from imbalanced data sets*. Cham: Springer, 2018.

[1] https://pan.webis.de/clef24/pan24-web/oppositional-thinking-analysis.html

- Rokach, L. (2019). *Ensemble learning: pattern classification using ensemble methods*. World Scientific Publishing Company Incorporated.
- Zhang, C., & Ma, Y. (Eds.). (2012). *Ensemble machine learning: methods and applications*. Springer Science & Business Media.
- Zhou, Z. H. (2012). *Ensemble methods: foundations and algorithms*. CRC press.
- *Seni, G., & Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. Synthesis lectures on data mining and knowledge discovery, 2(1), 1-126.*
- https://scikit-learn.org/stable/modules/svm.html
- https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifiier
- https://imbalanced-learn.org/stable/user_guide.html#user-guide
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

AUTOMATIC CLASSIFICATION (supervised models)

Supervised classification is a machine learning algorithm that aims to predict a categorical target variable based on a set of input features. The algorithm is trained on a labeled dataset, where the correct output is already known, to learn the relationship between the input features and the target variable. The trained algorithm can then predict the target

variable for new input data it has not seen before. The algorithm's accuracy[2] is measured by comparing the predicted output to the ground truth output for the test data. Some popular algorithms for supervised classification include support vector machines, logistic regression, decision trees, multilayer perceptrons, etc.

Support Vector Machines (SVMs)

SVMs are a type of machine learning algorithms used for classification and regression tasks[3]. The main idea behind SVMs is to find the best possible boundary or hyperplane to separate two or more classes in a high-dimensional space.

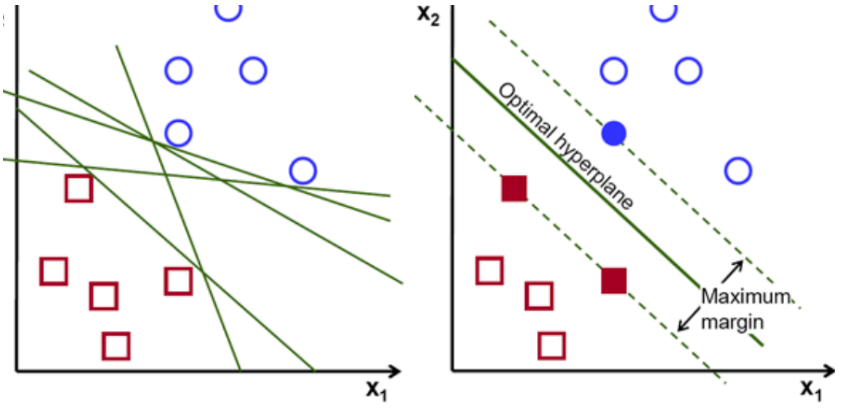

Figure 1. Hyperplane with maximum margin.

To do this, SVMs use a technique called *maximum margin classification*, which aims to find the hyperplane with the largest distance between the data points of the different classes (see the Figure 1). The data points closest to the hyperplane on either side are called *support vectors*, and they define the hyperplane.

[2] The accuracy is defined as the ratio of the number of correctly classified instances to the total number of instances in the dataset.

[3] The main difference between classification and regression tasks lies in the nature of the output variable. Classification tasks predict categorical variables, while regression tasks predict continuous variables.

SVMs work by mapping the input data points into a higher-dimensional space, where finding a hyperplane to separate the data points is more effortless. This is done using kernel functions, which calculate the similarity between data points in the high-dimensional space. Some common kernel functions include Linear, Polynomial, and Radial Basis Functions (RBF).

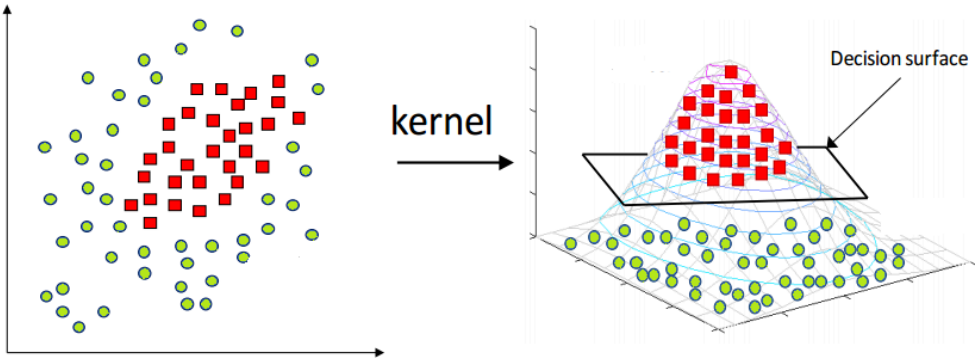

Figure 2. Transformation from a 2D to a 3D space.

Once the hyperplane is found, SVMs can classify new data points based on which side of the hyperplane they fall on. SVMs are helpful for image classification, spam detection, and text classification tasks such as sentiment analysis.

Advantages

- One of the advantages of SVMs is that they are less prone to overfitting than other classification algorithms, such as decision trees or neural networks.
- SVMs offer good accuracy in many problems of classification.
- SVMs use less memory than other classifiers because they use a subset of training points in decision-making.
- SVMs perform well with a clear separation margin and in high-dimensional spaces.

Disadvantages

- ➢ SVMs are not suitable for large datasets due to their high training time.
- ➢ SVMs are designed to solve binary classification problems.
- ➢ SVMs perform poorly with overlapping classes and are also sensitive to the choice of kernel function and the tuning of hyperparameters. Therefore, selecting these parameters to achieve the best performance is essential.

SVC, NuSVC and LinearSVC are methods to perform binary and multi-class classification on a dataset. SVC and NuSVC are similar but accept slightly different sets of parameters and have different mathematical formulations. On the other hand, LinearSVC is another implementation of Support Vector Classification for the case of a linear kernel. LinearSVC does not accept parameter kernels, which are assumed to be linear.

Hyperparameters

- ➢ *kernel*: Defines the transformation of the input data of the dataset.
- ➢ *regularization*: Reduces overfitting, this hyperparameter reduces the variance of the model parameters. However, it does so at the expense of adding bias to the estimation.
- ➢ *parameter C*: Controls how much error is tolerable in optimization. A lower value of C creates a hyperplane with a small margin, and a higher value of C makes a hyperplane with a more significant margin.

EXAMPLE 1

```
from sklearn.metrics import matthews_corrcoef
from sklearn.model_selection import train_test_split
from sklearn import svm
# Here, we are dividing the training set in two subsets one
for training and other for validation.
# In practice, X_en and X_es represent the matrices with text
representations from data provided, and Y_en and Y_es
represent the corresponding labels of either critical or
conspiracy texts.

X_en_train, X_en_test, y_en_train, y_en_test =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train, X_es_test, y_es_train, y_es_test =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

#Using the default parameters
clf_en = svm.SVC()
clf_en.fit(X_en_train, y_en_train)
predicted = clf_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

#Using the default parameters
clf_es = svm.SVC()
clf_es.fit(X_es_train, y_es_train)
predicted = clf_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

What can we do when we have more than two classes in our classification problems?

Multiclass classification problems are a type of machine learning problem where the goal is to classify data points into one of three or more predefined classes. In contrast to binary classification, where the goal is to classify data points in one of two classes, multiclass classification problems require classifiers to identify multiple possible outcomes. For this purpose, Sklearn implements two strategies (**One-vs-Rest** and **One-vs-One**) that allow extending the SVC operation to multiple classes.

- ➢ One-vs-Rest (OvR) approach: In this approach, SVM trains K binary classifiers, where K is the number of classes. Each classifier is trained to distinguish one class from the rest. During testing, each classifier produces a score for each class, and the class with the highest score is selected. This approach can be more efficient than One-v-One, but it may not work well when the classes are imbalanced. In the case of the SVC class, it is sufficient to define the parameter *decision_function_shape*; by default, it has the value *"ovr"* which represents the one-vs-rest approach.
- ➢ One-vs-One (OvO) approach: In this approach, the SVM trains K(K-1)/2 binary classifiers, where K is the number of classes. Each classifier is trained to distinguish between a pair of classes. Each classifier produces a prediction during testing, and the class with the most votes is selected. This approach works well when the number of classes is small. In case of a tie (between two classes with an equal number of votes), the class with the highest average value is selected. Since it requires adjusting K(K-1)/2 classifiers, this approach is usually slower than OvR, due to its quadratic complexity.

Logistic Regression (LR)

Logistic Regression is a binary classification algorithm that models the probability of a binary outcome (e.g., 0 or 1) as a function of one or more input variables. The model's output is a probability score, which is transformed into a binary prediction by applying a decision threshold. The algorithm estimates the coefficients of a linear function that map the input variables to the log-odds of the binary outcome. These coefficients are typically calculated using maximum likelihood estimation.

Advantages

- ➢ LR has several advantages, including its simplicity, interpretability, and ability to handle non-linear relationships between the input variables and the outcome.
- ➢ LR makes no assumptions about class distributions in feature space.
- ➢ LR performs well in many simple datasets, particularly, when the dataset is linearly separable.

Disadvantages

- ➢ LR assumes that the relationship between the input variables and the log-odds of the outcome is linear, and it may not work well when the classes are imbalanced or when the input variables are highly correlated.
- ➢ If the number of data points is smaller than the number of features, LR should not be used; otherwise, it may lead to overfitting.
- ➢ Non-linear problems cannot be solved with logistic regression because it has a linear decision surface.
- ➢ Linearly separable data are rarely encountered in real-world scenarios.

Hyperparameters

penalty: Specify the norm of the penalty by default the model uses l2 norm. Other options are *l1* and *elasticnet*[4].

tol: Tolerance for stopping criteria by default=1e-4

parameter C: Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

solver: Algorithm to use in the optimization problem. Default is *lbfgs*, the other possibilities are *liblinear*, *newton-cg*, *newton-cholesky*, *sag*[5] and *saga*[6].

EXAMPLE 2

```
from sklearn.metrics import matthews_corrcoef
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

X_en_train,      X_en_test,      y_en_train,      y_en_test      =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train,      X_es_test,      y_es_train,      y_es_test      =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

#Using the default parameters
clf_en = LogisticRegression()
clf_en.fit(X_en_train, y_en_train)
predicted = clf_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
```

[4] Elastic Net produces a regression model that is penalized with both the **L1-norm** and **L2-norm**. The consequence of this is to effectively shrink coefficients (like in ridge regression) and to set some coefficients to zero (as in LASSO).
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html

[5] Details about the mathematical formulation of the SAG method can be found in
https://link.springer.com/article/10.1007/s10107-016-1030-6

[6] Details about the mathematical formulation of the SAGA method can be found in
https://www.di.ens.fr/~fbach/Defazio_NIPS2014.pdf

```
print("MCC for English:", quality)

#Using the default parameters
clf_es= LogisticRegression()
clf_es.fit(X_es_train, y_es_train)
predicted = clf_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

Decision Trees (DTs)

Decision tree classifiers are a type of supervised machine learning algorithm that is used for both classification and regression problems. They work by recursively partitioning the input space into regions, each associated with a class label or a predicted value. The algorithm builds a tree-like model where each internal node represents a test on an input variable, and each branch corresponds to the possible outcomes of the test.

Advantages

- ➢ DTs have several advantages, including their interpretability and simplicity. They are used as white box models. Trees can be visualized.
- ➢ DTs have the ability to handle numerical and categorical input variables. Notice that the Sklearn implementation does not support categorical variables.
- ➢ DTs capture non-linear relationships between the input variables and the outcome.
- ➢ DTs can handle missing and noisy data and are robust to outliers.

Disadvantages

- ➢ DTs are prone to overfitting, especially when the tree is deep and complex. Several techniques have been developed to mitigate overfitting, including pruning, ensemble methods (such as random forests and gradient boosting), and regularization.
- ➢ DTs can be unstable because slight variations in the data can generate a completely different tree.
- ➢ DTs learning algorithms rely on heuristic algorithms to make locally optimal decisions at each node. Such algorithms cannot guarantee a globally optimal decision tree.

Hyperparameters

criterion: The function to measure the quality of a split. Supported criteria are *gini*, *entropy*, *log_loss* by default, the criterion used is *gini*[7].

splitter: The strategy used to choose the split at each node. Supported strategies are *best* to choose the best partition and *random* to choose the best random split.

max_depth: The maximum depth of the tree. If *None*, nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples.

min_samples_split: The minimum number of samples required to split an internal node, the default value is set to 2.

min_samples_leaf : The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least *min_samples_leaf* training samples in each left and right branch. This may have the effect of smoothing the model, especially in regression.

[7] The mathematical formulation for the gini criterion can be found in the link https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation

max_features: The number of features to consider when looking for the best split. It can be set to an integer value, float value or *auto*, *sqrt*, and *log2, by default it is set to None.*

EXAMPLE 3

```
from sklearn.metrics import matthews_corrcoef
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

X_en_train,     X_en_test,     y_en_train,     y_en_test     =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train,     X_es_test,     y_es_train,     y_es_test     =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

clf_en = DecisionTreeClassifier()
clf_en.fit(X_en_train, y_en_train)
predicted = clf_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

clf_es = DecisionTreeClassifier()
clf_es.fit(X_es_train, y_es_train)
predicted = clf_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) is a type of Artificial Neural Network (ANN) that consists of multiple layers of interconnected nodes (or neurons) that process and transform input data into output. The input

data is fed into the input layer, which passes the data through one or more hidden layers before producing the outcome in the final output layer. Each neuron in the MLP receives input from the neurons in the previous layer, calculates a weighted sum of the inputs, applies an activation function, and passes the output to the next layer. The weights and biases of the neurons are learned through a training process that adjusts them to minimize the error between the predicted output and the true output. MLPs are commonly used in classification and regression tasks and are prevalent in applications such as image recognition, speech recognition, and natural language processing.

Advantages

- ➢ MLPs are capable of learning non-linear relationships between inputs and outputs, making them useful for complex problems where linear models may not be sufficient.
- ➢ MLPs can be used for both classification and regression tasks and can handle a variety of input.
- ➢ MLPs can handle large amounts of data and can be scaled up by adding more layers or neurons, allowing them to handle more complex tasks.
- ➢ MLPs are capable of generalizing from training data to unseen data, making them suitable for tasks such as pattern recognition and prediction.
- ➢ MLPs can adapt to changing inputs and can learn from new data without requiring retraining from scratch.
- ➢ MLPs can be trained and run on parallel computing systems, allowing for faster processing of large amounts of data.

Disadvantages

- ➢ MLPs can take a long time to train, especially for large and complex datasets. The training process can be computationally expensive and may require a lot of resources.
- ➢ MLPs can be prone to overfitting, where the model learns to fit the training data too closely and does not generalize well to new data. Regularization techniques can be used to address this issue.
- ➢ MLPs can get stuck in local optima during the training process, where the model converges to a suboptimal solution instead of the global optimum. This can be addressed by using various optimization algorithms and techniques.
- ➢ MLPs can be difficult to interpret and understand, making it challenging to explain how the model makes predictions.

Hyperparameters

hidden_layer_sizes : array-like of shape(*n_layers - 2*,), *default=(100,).* The *ith*-element represents the number of neurons in the *ith*-hidden layer.

activation : Activation function for the hidden layer, *default=relu*. Other activation functions are *identity*, *logistic*, *tanh* and *relu*.

solver : The solver for weight optimization, *default=adam*. Other solvers are *lbfgs* and *sgd.*

batch_size: Size of minibatches for stochastic optimizers, *default=auto*. If the solver is *lbfgs*, the classifier will not use minibatch. When set to *auto*, *batch_size=min(200, n_samples)*.

learning_rate: Learning rate schedule for weight updates, *default=constant*. Other schedules are *invscaling* and *adaptive.*

learning_rate_init: The initial learning rate used, *default=0.001*. It controls the step-size in updating the weights. Only used when solver=*sgd* or *adam*.

max_iter: Maximum number of iterations, *default=200*. The solver iterates until convergence. For stochastic solvers (*sgd* and *adam*), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.
tol: Tolerance for the optimization, *default=1e-4*. When the loss or score is not improving by at least *tol* for *n_iter_no_change* consecutive iterations, unless *learning_rate* is set to adaptive, convergence is considered to be reached and training stops.
early_stopping: Whether to use early stopping to terminate training when validation score is not improving, default=False. If set to True, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least *tol* for *n_iter_no_change* consecutive epochs. The split is stratified, except in a multilabel setting. If early stopping is False, then the training stops when the training loss does not improve by more than tol for *n_iter_no_change* consecutive passes over the training set. It is only effective when solver=*sgd* or *adam*.
validation_fraction: The proportion of training data to set aside as validation set for early stopping, *default=0.1*. Must be between 0 and 1. Only used if *early_stopping* is True.
n_iter_no_change: Maximum number of epochs to not meet *tol* improvement, *default=10*. Only effective when solver=*sgd* or *adam*.

EXAMPLE 4

```
from sklearn.metrics import matthews_corrcoef
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

X_en_train, X_en_test, y_en_train, y_en_test = train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train, X_es_test, y_es_train, y_es_test = train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)
```

```
#Using the default parameters
clf_en = MLPClassifier(random_state=1, max_iter=300)
clf_en.fit(X_en_train, y_en_train)
predicted = clf_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

#Using the default parameters
clf_es = MLPClassifier(random_state=1, max_iter=300)
clf_es.fit(X_es_train, y_es_train)
predicted = clf_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

ENSEMBLE OF MODELS

Ensemble methods can often outperform simple classifiers due to the concept of the *"wisdom of the crowd"* or *"collective intelligence".* By combining the predictions of multiple models ensemble methods can mitigate the individual weaknesses of each model and produce more accurate and robust predictions. One theoretical foundation for why ensemble methods work well is the *"bias-variance tradeoff"*. In machine learning, the goal is to build a model that can accurately generalize to new data points that have not been seen before. The *“bias-variance tradeoff”* refers to the tradeoff between a model's ability to fit the training data (low bias) and generalize on new data (low variance).

Simple models such as decision trees tend to have high bias and low variance, meaning they are likely to underfit the training data but have low variance in their predictions. On the other hand, complex models such as neural networks tend to have low bias but high variance, meaning that they are likely to overfit the training data and have high variance in their predictions. Ensemble methods can help to balance the "bias-variance tradeoff" by combining multiple models with different biases and variances. For example, bagging and random forest methods use multiple models to create an ensemble with lower variance and better generalization performance. Boosting techniques such as AdaBoost and Gradient Boosting can improve bias and variance by iteratively adding new models that focus on correcting the errors of previous models.

Approaches to combine models

The most common approaches for supervised ensemble models in machine learning are:

1. **Voting**: In this approach, multiple base models are trained on the same dataset, and the final prediction is made by taking a majority vote of the predictions of the single models. This approach is typically used when the base models are diverse and complementary.
2. **Bagging**: In this approach, multiple instances of the same base model are trained on different subsets of the training data. The final prediction is made by averaging the predictions of the individual models. Bagging is typically used when the base model has high variance or is unstable.
3. **Boosting**: In this approach, multiple base models are trained sequentially, with each subsequent model learning from the errors of the previous model. The final prediction is made by taking a weighted average of the predictions of the single models. Boosting

is typically used when the base model is weak, aiming to improve accuracy and reduce bias.

4. **Stacking**: In this approach, multiple base models are trained on the same dataset, and the predictions of the individual models are used as input features for a final model. The final model learns to combine the predictions of the single models to improve accuracy. Stacking is typically used when the base models are diverse and complementary.

VOTING

The idea is that by combining the predictions of multiple models that are trained on the same data but using different algorithms or parameter settings, the resulting prediction will be more accurate and robust than any single model's prediction.

There are two main types of voting ensembles: *“hard voting”* and *“soft voting”*. In hard voting, the prediction of the ensemble is determined by a simple majority vote of the predictions of the base models. For example, if three base models predict that an input belongs to the class *humor*, and two predict that it belongs to the class *prejudiced humor*, then the ensemble would predict the class *humor*. In soft voting, the ensemble prediction is based on the average probabilities of the base models' predictions for each class. The class with the highest average probability is then selected as the final prediction.

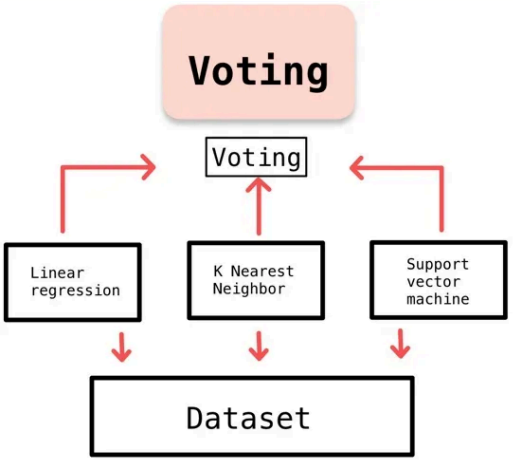

Figure 3. Voting ensemble: hard voting

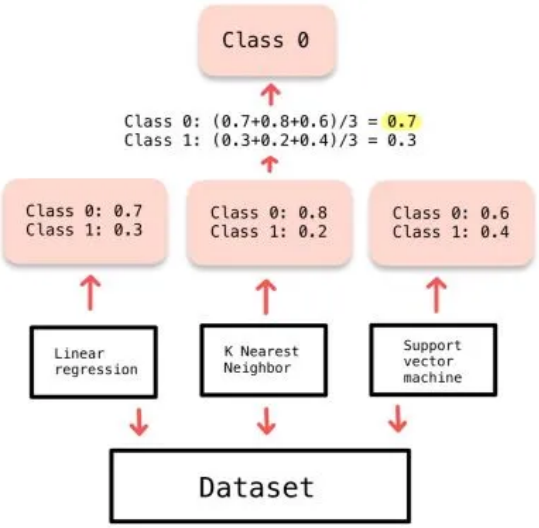

Figure 4. Voting ensemble: soft voting

Voting ensembles can be applied to a wide range of machine learning tasks, including classification, regression, etc. They can be particularly useful in situations where the performance of a single model is limited, or where different models have different strengths and weaknesses. By combining the predictions of multiple models, voting ensembles can provide a more accurate and robust prediction that is less likely to be affected by individual model biases or errors.

EXAMPLE 5

```
from sklearn.metrics import matthews_corrcoef
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

X_en_train, X_en_test, y_en_train, y_en_test =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train, X_es_test, y_es_train, y_es_test =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

# create the base models, Logistic Regression, Decision Tree,
and Support Vector Machine
model1 = LogisticRegression()
model2 = DecisionTreeClassifier()
model3 = SVC()
# create the voting ensemble
ensemble_en = VotingClassifier(estimators=[('lr', model1),
('dt', model2), ('svm', model3)], voting='hard')
# train the ensemble
ensemble_en.fit(X_en_train, y_en_train)
# Evaluate the ensemble
predicted=ensemble_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

# create the base models, Logistic Regression, Decision Tree,
and Support Vector Machine
model1 = LogisticRegression()
model2 = DecisionTreeClassifier()
model3 = SVC()
# create the voting ensemble
```

```
ensemble_es = VotingClassifier(estimators=[('lr', model1),
('dt', model2), ('svm', model3)], voting='hard')
# train the ensemble
ensemble_es.fit(X_es_train, y_es_train)
# Evaluate the ensemble
predicted=ensemble_en.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

The voting parameter is set to *'hard'* to indicate that we want to use hard voting.

BAGGING

Bagging is based on bootstrap resampling, which involves randomly sampling the training data with replacement to create multiple subsets of the same size as the original data set. The basic idea behind bagging is to train a set of base models on different bootstrap samples of the training data and then aggregate their predictions using a simple averaging or voting scheme to create the final prediction. This can help reduce the model's variance and improve its generalization performance.

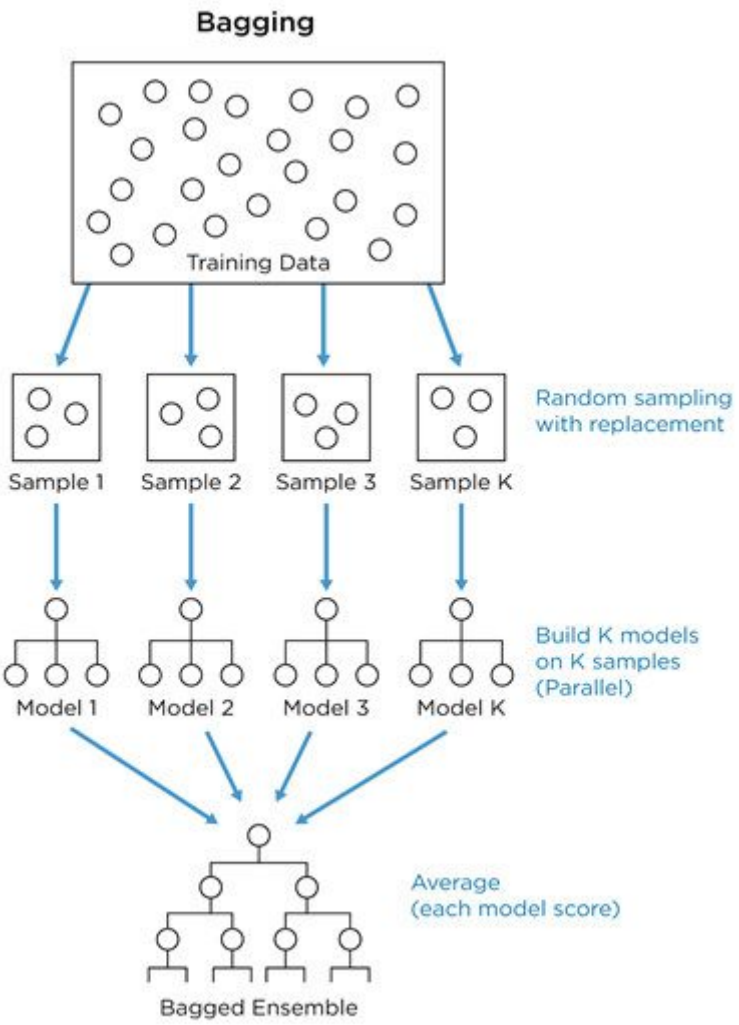

Figure 5. General schema of the bagging ensemble

The bagging algorithm can be summarized as follows:

1. Create multiple bootstrap samples of the training data, each of the same size as the original data set.
2. Train a base model on each bootstrap sample.
3. Aggregate the predictions of the base models using a simple averaging or voting scheme to create the final prediction.

Random Forest is an extension over bagging. In this approach, multiple decision trees are trained on different subsets of the training data. It

also takes the random selection of features rather than using all features to grow trees. The final prediction is made by averaging the predictions of the individual trees. Random forests are typically used when the dataset has high dimensionality and complex relations among the features.

EXAMPLE 6

```
from sklearn.metrics import matthews_corrcoef
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

X_en_train,     X_en_test,     y_en_train,     y_en_test     =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train,     X_es_test,     y_es_train,     y_es_test     =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

ensemble_en
=BaggingClassifier(base_estimator=DecisionTreeClassifier(),
n_estimators=10)
# train the ensemble
ensemble_en.fit(X_en_train, y_en_train)
# evaluate the ensemble
ensemble_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

ensemble_es
=BaggingClassifier(base_estimator=DecisionTreeClassifier(),
n_estimators=10)
# train the ensemble
ensemble_es.fit(X_es_train, y_es_train)
# evaluate the ensemble
ensemble_es.predict(X_es_test)
```

```
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

EXAMPLE 7

```
from sklearn.metrics import matthews_corrcoef
from sklearn.ensemble import RandomForestClassifier

X_en_train,     X_en_test,     y_en_train,     y_en_test     =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train,     X_es_test,     y_es_train,     y_es_test     =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

ensemble_en           =RandomForestClassifier(n_estimators=100,
max_depth=10, random_state=0)
# train the ensemble
ensemble_en.fit(X_en_train, y_en_train)
# evaluate the ensemble
ensemble_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

ensemble_es           =RandomForestClassifier(n_estimators=100,
max_depth=10, random_state=0)
# train the ensemble
ensemble_es.fit(X_es_train, y_es_train)
# evaluate the ensemble
ensemble_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

BOOSTING

Boosting is an ensemble learning method combining multiple weak learners to create a strong learner. Conversely to bagging, which trains each weak learner independently, boosting trains weak learners sequentially, with each subsequent learner concentrating on the data points the previous learners misclassified. For that, the boosting method reweights the training data iteratively so that the next model focuses more on the data points that the previous model misclassified. AdaBoost and XGBoost are two boosting approaches.

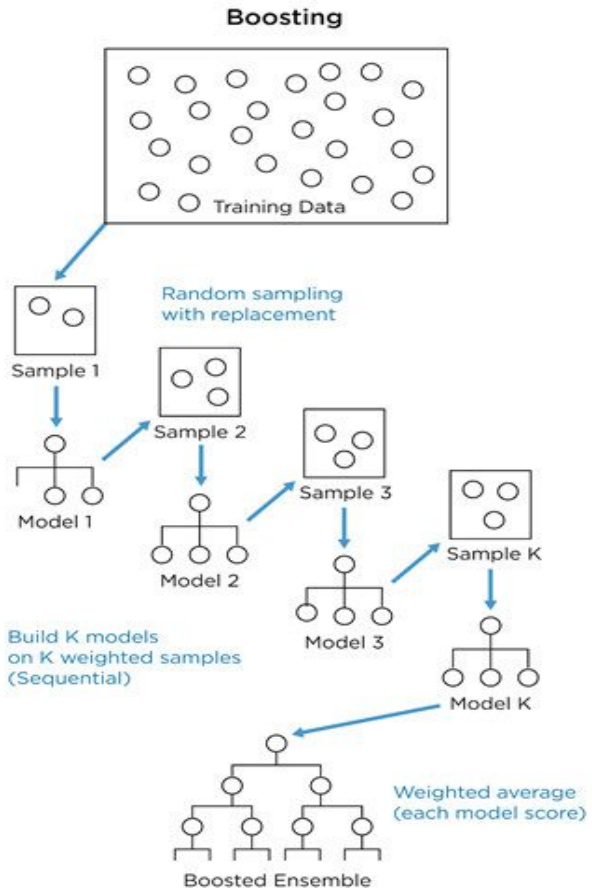

Figure 6. General schema of the boosting ensemble

AdaBoost (Adaptive Boosting) is a popular boosting algorithm[8] introduced by Freund and Schapire in 1997. The AdaBoost algorithm works by iteratively adding weak learners to strong learners. At each iteration, the algorithm selects a weak learner that can best classify the examples that the previous weak learners misclassified. The final model is a weighted combination of the weak learners, where the weights are determined by the accuracy of each weak learner on the training data.

The mathematical foundation of AdaBoost can be described as follows. Suppose we have a dataset with input features *X* and corresponding target values *y*, where y is a binary classification label (+1 or -1). We also have a set of base models, denoted by *M={Model1, Model2, ..., Modelk}*, which can be used to make binary predictions on X. AdaBoost aims to train a weighted combination of the base models that can make accurate binary predictions on *X*.

EXAMPLE 8

```
from sklearn.metrics import matthews_corrcoef
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression

X_en_train, X_en_test, y_en_train, y_en_test =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train, X_es_test, y_es_train, y_es_test =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

# Create the boosting classifier
ensemble_en=AdaBoostClassifier(base_estimator=LogisticRegressi
on(),n_estimators=10)
# Train the boosting classifier on the training data
ensemble_en.fit(X_en_train, y_en_train)
```

[8] Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119-139.

```
# Evaluate the boosting classifier on the testing data
predicted = ensemble_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

# Create the boosting classifier
ensemble_es=AdaBoostClassifier(base_estimator=LogisticRegressi
on(),n_estimators=10)
# Train the boosting classifier on the training data
ensemble_es.fit(X_es_train, y_es_train)
# Evaluate the boosting classifier on the testing data
predicted = ensemble_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

XGBoost (eXtreme Gradient Boosting)[9] is a robust ensemble learning algorithm introduced by Chen and Guestrin in 2016. It extends the gradient boosting method, focusing on scalability and speed. XGBoost is widely used in machine learning competitions and is known for its high accuracy and performance. The XGBoost method minimizes a differentiable loss function, such as mean squared error or log loss, using gradient descent. XGBoost iteratively adds decision trees to a weighted combination of previous trees, where the weights are determined by the negative gradient of the loss function.

[9] Chen, T., & Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (pp. 785-794).

EXAMPLE 9

```
!pip install xgboost
import xgboost as xgb
from sklearn.metrics import matthews_corrcoef


X_en_train,     X_en_test,     y_en_train,     y_en_test     =
train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train,     X_es_test,     y_es_train,     y_es_test     =
train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

# Convert data to DMatrix format
dtrain_en = xgb.DMatrix(X_en_train, label=y_en_train)
dtest_en = xgb.DMatrix(X_en_test, label=y_en_test)

# Set XGBoost parameters
params = {
  'max_depth': 3,
  'eta': 0.1,
  'objective': 'binary:logistic',
  'eval_metric': 'auc'
}

# Train the model
num_rounds = 100
bst_en = xgb.train(params, dtrain_en, num_rounds)
# Make predictions on the test set
predicted = bst_en.predict(dtest_en)
# Evaluate the model
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
predicted = np.asarray([round(value) for value in predicted])
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)
```

```
# Convert data to DMatrix format
dtrain_es = xgb.DMatrix(X_es_train, label=y_es_train)
dtest_es = xgb.DMatrix(X_es_test, label=y_es_test)
# Set XGBoost parameters
params = {
  'max_depth': 3,
  'eta': 0.1,
  'objective': 'binary:logistic',
  'eval_metric': 'auc'
}
# Train the model
num_rounds = 100
bst_es = xgb.train(params, dtrain_es, num_rounds)
# Make predictions on the test set
predicted = bst_es.predict(dtest_es)
# Evaluate the model
print("Predicted\t-->\t", predicted)
predicted = np.asarray([round(value) for value in predicted])
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

STACKING

Stacking, or stacked generalization, is an ensemble learning technique combining multiple base models to improve prediction performance. The basic idea behind stacking is to train a meta-model on the predictions of several base models, hoping that the meta-model can learn to combine the strengths of the base models and mitigate their weaknesses.

The mathematical foundations of stacking can be described as follows. Suppose we have a dataset with input features *X* and corresponding labels values *y*. We have a set of base models, denoted by *M={Model1, Model2, ..., Modelk}*, which can be used to make predictions on *X*.

Stacking aims to train a **meta-model**, denoted by *M'*, that can make more accurate predictions on *X* than each individual base model in *M*.

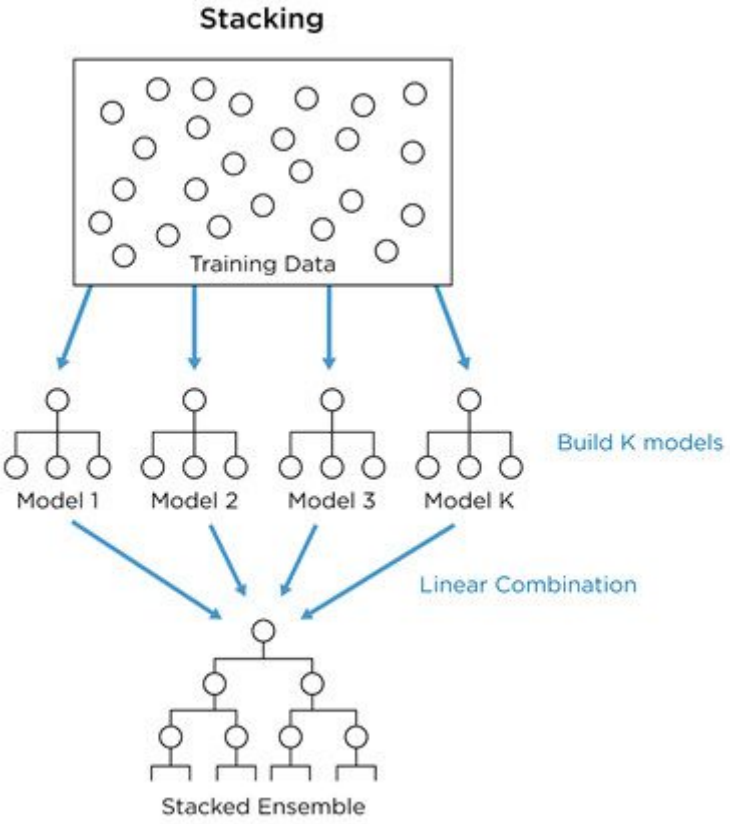

Figure 7. General schema of the stacking ensemble

Firstly, we first split the data into training and testing sets to do this. We then use the training set to train the base models in *M* and use them to make predictions on the testing set. We stack these predictions together to form a new feature matrix, denoted by *Z*, where each row corresponds to a sample in the testing set, and each column corresponds to the predicted value of the test set by one of the base models in *M*. Then, we use *Z* as the new input features to train *M'*. The target values for *M'* are the true label values of the testing set. *M'* can be any model type, such as linear regression, logistic regression, or neural network. Once *M'* has been trained, we can predict new data points.

Stacking is a robust technique for improving prediction performance in machine learning, but it should be used wisely and carefully considering the computational and interpretational costs.

EXAMPLE 10

```
from sklearn.metrics import matthews_corrcoef
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split


X_en_train, X_en_test, y_en_train, y_en_test = train_test_split(X_en, Y_en, test_size=0.1, random_state=1234)
X_es_train, X_es_test, y_es_train, y_es_test = train_test_split(X_es, Y_es, test_size=0.1, random_state=1234)

# create the base models,K-NNeighbors, Logistic Regression, Decision Tree, and Support Vector Machine
base_models_en = [('dt', DecisionTreeClassifier()), ('svm', SVC()), ('knn', KNeighborsClassifier())]
# Create the meta-model
meta_model_en = LogisticRegression()
# Create the stacking classifier
ensemble_en = StackingClassifier(estimators=base_models_en, final_estimator=meta_model_en)
# Train the stacking classifier on the training data
ensemble_en.fit(X_en_train, y_en_train)
# Evaluate the stacking classifier on the testing data
predicted = ensemble_en.predict(X_en_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_en_test)
```

```
quality=matthews_corrcoef(y_en_test, predicted)
print("MCC for English:", quality)

base_models_es = [('dt', DecisionTreeClassifier()), ('svm',
SVC()), ('knn', KNeighborsClassifier())]
# Create the meta-model
meta_model_es = LogisticRegression()
# Create the stacking classifier
ensemble_es = StackingClassifier(estimators=base_models_es,
final_estimator=meta_model_es)
# Train the stacking classifier on the training data
ensemble_es.fit(X_es_train, y_es_train)
# Evaluate the stacking classifier on the testing data
predicted = ensemble_es.predict(X_es_test)
print("Predicted\t-->\t", predicted)
print("GroundTruth\t-->\t", y_es_test)
quality=matthews_corrcoef(y_es_test, predicted)
print("MCC for Spanish:", quality)
```

HYPERPARAMETERS SETTING

Hyperparameters are parameters that are not learned directly within the classifier. It is possible and recommended to search the hyperparameter space for better results. Two generic approaches can be used for parameter search:

Grid Search: exhaustively considers all parameter combinations.

EXAMPLE 11

```
from sklearn import svm
from sklearn.metrics import matthews_corrcoef, make_scorer
mcc_scorer = make_scorer(matthews_corrcoef,
sample_weight=None)
from sklearn.model_selection import GridSearchCV
```

```
clf = GridSearchCV(estimator=svm.SVC(), param_grid={'C': [2,3,4,5,10], 'kernel': ('linear', 'rbf')}, scoring=mcc_scorer, n_jobs=-1, verbose=1)
clf.fit(X_en, Y_en)
#Obtaining the best hyper-parameters
print(clf.best_params_)
#Obtain the best classifier
print(clf.best_estimator_)
```

Randomized Search: samples a parameter space with a specific distribution.

EXAMPLE 12

```
#- uniform: uniform distribution in: [loc, loc + scale]
#- uniform: uniform distribution in: [loc, loc + scale]
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import matthews_corrcoef, make_scorer
mcc_scorer = make_scorer(matthews_corrcoef, sample_weight=None)
from scipy.stats import uniform
logistic = LogisticRegression()
distributions = dict(C=uniform(loc=0, scale=4), penalty=['l2', 'l1'])
clf = RandomizedSearchCV(logistic, distributions, scoring=mcc_scorer, n_jobs=-1, verbose=1)
clf.fit(X_es, Y_es)
print(clf.best_params_)
print(clf.best_estimator_)
```

GridSearchCV and RandomSearchCV have successive equivalents for halving the search: HalvingGridSearchCV and HalvingRandomSearchCV.

ACTIVITY

Students must train and validate machine learning models for the "*Oppositional thinking analysis: Conspiracy theories vs. critical thinking narratives*" task. The goal is to distinguish conspiracy narratives from oppositional narratives that do not express a conspiracy mentality (i.e., critical thinking), a binary classification problem. For that purpose:

1. Students must train and evaluate **at least four classifiers** and compare their results. One of the classifiers must be an ensemble of three simple models.
2. For each of the three groups of text representation studied in the previous practice (traditional forms, static embedding-based, and contextual embedding-based), students have to choose one representation method and train the selected models using that representation. Preprocess the text as deemed appropriate for each representation method.
3. Students must train and evaluate models for both Spanish and English datasets.

The proposed models' performance must be evaluated based on the official evaluation metric (Matthew correlation coefficient) provided by the task organizers. The official training dataset for both languages are in the file **Dataset-Oppositional.zip**

The **source code** and a **short report** explaining the text representations, the classification models, the hyperparameters considered, and the results obtained should be delivered before the next session.